PBT Framework and BT Reductions

May 2008

Nassim Nasser

Supervisor: Jeff Edmonds

Abstract

Borodin et. al. [1] and [2] proposed a model that captures dynamic programming and backtracking algorithms (pBT). This model is useful in measuring the optimality of dynamic programming and backtracking algorithms. They also defined a reduction strategy which is specific to this model, called BT-reduction. In this project, we will present the pBT model as it appeared in [1] and [2] and give a direction to further extend this model. We present some reductions to well-known NP-complete problems and show whether they fit within the model through BT-reduction.

Contents

1. Introduction

There is a vast number of computational problems and there exists many algorithms to address them. Yet most of the efficient algorithms for solving these problems fit into a small set of design paradigms such as hill climbing, divide and conquer, greedy algorithms and dynamic programming.

Recently, formalizing these design paradigms has received considerable attention. In [4] a formal framework of priority algorithms for scheduling problems was developed which captures greedy algorithms. This model was further extended in [3] to capture facility location and set cover. Further extension of this model [6] captures graph problems. In [1] and [2] a formal framework for dynamic programming and backtracking was introduced. This model allows us to evaluate simple dynamic programming algorithms. An extension to this model was provided in [7], which formalizes the notion of memoization. This was further extended to a more flexible model in [8].

Having these formalized definitions is useful because they let us analyze the power and limitations of a large class of algorithms. These frameworks enable us to derive non-trivial lower bounds on the performance of a particular class of algorithms (e.g. Greedy algorithms) with respect to how close to optimal a solution the algorithm can find.

The study of lower bounds is useful because it establishes a strong (negative) result about the achievable performance of a large class of algorithms. When the lower bound is tight, any algorithm achieving that ratio is optimal for the given class. Lower bounds can help us in designing algorithms, since they indicate where improvements are possible and where they are not. They also serve as an evaluation of the power of an algorithmic paradigm.

In this project, we worked on the backtracking and dynamic programming model proposed in [1]. The main focus of this project was to find problems that fit the model by using a particular type of reduction, specific to the model, called BT-reduction. Investigating in this direction is useful because it shows the power and extensiveness of the model in terms of its ability to capture various dynamic programming algorithms. Also, by these reductions, we can give lower bounds for problems within this framework.

The remainder of this paper is organized in 5 sections. First, in Section 2, we present the definition of dynamic programming in general. Then, in Section 3, we introduce the model for dynamic programming and backtracking algorithms, pBT, as it appeared in [1] and [2] and summarize the results from that work. We also consider an extension to this model from [8]. We present an extension of our own to this model, which increases, we believe, the model's expressiveness, thus covering a wider range of algorithms. In Section 4, we define the BT-reduction. In Section 5, we present some reductions to well-known NP-complete problems and show whether they fit within the model. We end this report with a conclusion and guidelines for future works in Section 6.

2. Dynamic Programming [9]

One method to solve optimization problems is dynamic programming. It provides polynomial time algorithms for many important and practical problems. The key component in dynamic programming is the recurrence relation. This relation tells us how to build an optimal solution for the overall problem by using optimal solutions for smaller instances of the same problem. For example, to find the shortest path to the sink node in a directed, leveled graph, we can first compute the shortest path to the sink from all adjacent vertices, and then use this, recursively, to find the best overall path. To generalize, first we divide the overall instance to smaller instances (sub-problems), then find the optimal solution for these smaller instances and afterwards use these partial optimal solutions to find the optimal solution for the overall instance. Note that each of these smaller instances are, themselves, solved by dividing them into smaller instances (sub-subproblems), until we reach some simple case that is solvable in constant time.

Using just this recurrence relation, we might recurse on some particular sub-instances many times. It would be wise to store the solution for each sub-instance in a table, so that the next time we see the sub-instance, we can simply use the stored solution. This saves us time, since we do not need to recompute the solutions. This approach is called memoization. It is also possible sometimes to prune some solutions to save space. If we know that we will not need a particular solution we can remove it from our table.

3. A Model for Backtracking and Dynamic Programming (pBT)[1] and [2]

This model is based on the priority model proposed in [4]. The priority model captures greedy algorithms. A major difference between this model (pBT) and the priority model is that, in the latter, the algorithm considers only one solution, whereas in this model the algorithm is able to maintain different partial solutions in parallel. This is done by using a branching tree, which is referred to as the computation tree. The width of this tree, the maximum number of nodes over all depth levels, is important for two reasons. First, the width of the tree times its depth gives a universal upper bound on the running time of any search style algorithm. Second, it identifies the maximum number of partial solutions that must be kept in parallel during the computation.

For this model, the input is presented as a set of data items. Each data item gives us a small amount of information about the problem. For example, for the 3-SAT problem a data item represents a propositional variable with all the clauses it is in. For the Knapsack problem, a data item is an object that has a weight and a value.

An optimization problem, P, is represented by a pair $(D, \{f_p^n\})$ where D is the domain of the data items and $\{f_p^n\}$ is a set of objective functions, for each input size n, that needs to be optimized. The objective function $f_p^n : (D_{1,} D_{2,} \ldots, D_n, a_{1,} a_{2,} \ldots, a_n) \rightarrow R$ takes as input a set of data items (D_1 , D_2, ... , D_n) and a set (a_1, a_2, ... , a_n) of decisions made about those data items and returns a value. This value, depending on the optimization problem, needs to either minimized or maximized. To make the definition more robust, we also include a set of possible decisions H, such that each a_i is included in H. For example, in the Knapsack problem the set of data items are represented as a pair <weight,value> and set of allowable choices is {reject, accept, abort}. The abort decision is for when we know that following a particular path in the computation tree is no longer feasible.

Now that we have defined our problem, let us see how the model works. The algorithm has an ordering function that determines in what order the data items should be considered. Based on how the ordering is done, we can define three types of algorithms: fixed order, adaptive and fully adaptive.

If we have a *fixed order* algorithm, then our algorithm first sorts the data items according to some criterion. Then it visits the data items based on that ordering and makes decisions based on them. The ordering of the data items in this case never changes over the course of the algorithm. For instance, a fixed order Knapsack algorithm might sort the input objects based on their value to weight ratio and then follow that ordering to consider each data item in turn.

A more general strategy is to allow our algorithm to re-sort the data items based on the items seen so far (*adaptive algorithm*). As an example we can think of the interval cover problem. For this problem we have a set of points and set of intervals. We want to find a minimum subset of (possibly overlapping) intervals that covers all the points. Here we can assume that we have fixed set of sorted points <P> and our data items are the intervals. Each interval has a start and end point. For example <I_1, S_1, F_1> means that interval I_1 starts at point S_1 and ends at point F_1. Figure 1 illustrates an instance of the interval cover problem and one possible optimal solution. [9]

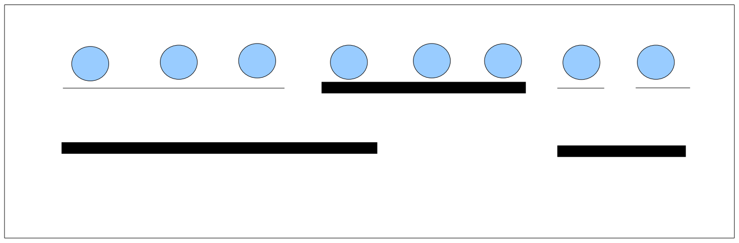

Figure 1. An instance for the interval cover problem. One possible solution shown with thick lines.

In one approach, the algorithm sorts the intervals by end time. It looks at the left-most uncovered point P_i and accepts the interval that covers this point and extends furthest to the right. After covering more points, depending on the left-most uncovered point, this ordering changes.

To make the backtracking and dynamic programming model even more powerful, we can make the following extension. We can allow an algorithm to re-sort the remaining data items not only based on the items seen so far, but also based on the decisions made on those items. This is useful when the algorithm can make multiple decisions on the same object. This will give us a *fully adaptive* algorithm. This allows the model to capture DPLL[1] algorithms for SAT.

The general idea behind this model is that, at first, the data items are an unknown set, and

1 The **DPLL (Davis-Putnam-Logemann-Loveland)** algorithm is a backtracking algorithm for deciding the satisfiability of propositional logic formula in conjunctive normal form.

that the algorithm has an ordering function that orders the data items. At the end of each run, the algorithm has seen a subset of the data items ($D_1, ..., D_k$) and has made some decisions on them ($a_1, ..., a_k$). At the next run, depending on the type of the algorithm (fixed, adaptive or fully adaptive), the algorithm might be able to re-order the remaining items. The next new item that is revealed to the algorithm is the first one in the ordered list. At this point, the algorithm may decide to make a single decision on the data item, or may choose to make multiple decisions. Trying multiple decisions is done by branching on that data item in the computation tree. Each node of the computation tree is labeled with the revealed data item, D_k, and each edge is labeled with a decision, a_k, made on that item. In this way, each path in the computation tree stores a partial solution for the part of the input that has been seen so far. A key aspect of this model is that when the algorithm makes a decision about the new data item, this decision must extend an existing path in the tree. This ensures that each new partial solution will be the extension of some existing ones. The algorithm continues seeing new data items and making decision(s) on them until all the data items have been revealed.

There is also an equivalent and, perhaps, easier way of understanding the algorithm. Instead of considering the computation tree, we can say that the algorithm saves a set of possible partial solutions, {<a_1, ... , a_{k-1}>, <a'_1, ... , a'_{k-1}> , ...}, for the items seen so far <D_1, ..., D_{k-1}>. After seeing the new data item, D_k, the algorithm can extend some of these partial solutions with the decisions made about D_k. The algorithm can also throw some partial solutions away by not extending them.

It is also worth mentioning how the different sorting strategies (fixed, adaptive, fully adaptive) affects the computation tree. For the fixed order algorithm the data items will be sorted once and the ordering of the items does not change throughout the construction of the tree. The adaptive algorithm can re-order the remaining data items based on the data items seen so far <D_1, ..., D_k>. However, this ordering does not depend on the decisions made on the revealed data items, therefore each path of the tree will use the same ordering and each node at level K of the tree will be labeled with the same data item, D_k.

In the fully adaptive algorithm the ordering is affected by both the data items seen so far and the decisions made on them. This means the ordering can change at every branch of the tree and hence each path of the computation tree might have seen different data items.

A formal definition of the backtracking algorithm and the computation is as follows.

Definition 1. As appeared in [1] and [2]. A backtracking algorithm A for problem

$P = (D, \{f_p^n\})$ consists of the ordering functions

$$r_A^k : D^k \times H^k \rightarrow O(D)$$

and the choice functions

$$c_A^k : D^{k+1} \times H^k \rightarrow 2^H,$$

or equivalently

$$c_A^k : D^{k+1} \times H^k \rightarrow O(H \cup \{\perp\})$$

where $\{\perp\}$ is the choice to abort

We have the following classes of algorithms:

1. Fixed algorithms: the ordering functions do not depend on its arguments.
2. Adaptive algorithms: the ordering functions only depend on D^i s.
3. Fully adaptive algorithms: the ordering functions depend on both D^i s and H^i s.

In the above definition *O(D)* and *O(H* ∪ {⊥}*)* indicate the set of all orderings of *D* and *H* ∪ {⊥} respectively and 2^H indicates the set of subsets of H.

Definition 2. Assume that P is a search/optimization problem, $P = (D, \{f_p^n\})$, and A is a backtracking algorithm for P. For any instance I = (D_1, ..., D_n), $D_i \in D$ we define the computation tree T_A(I) as an oriented tree recursively, as follows.

- Each node v of depth k in the tree is labeled with a data item, $D_{k'}$. Each edge is labeled with a decision, $a_{k'}$.
- A new node v is formed in the tree as follows. The new node may either be the root or be the extension of some already existing edge e of the tree. The nodes along the path from the root to the current edge e is labeled with the data items $<D_1, ..., D_k>$ seen so far and the edges extending these nodes are labeled with the decisions $<a_1, ..., a_k>$ made about them. Given this information, the algorithm uses the ordering function $r_A^k : D^k \times H^k \rightarrow O(D)$ to provide an order $r_A^k(D_{1,}..., D_k, a_{1,}..., a_n)$ of the set of possible future data items in D. The new data item D_{k+1} is the first unseen item from input instance according to this ordering function. The end point node v of e will be labeled by this new data item.
- An edge e' is formed to extend node v in the tree as follows. Again, the path up to node v is labeled with the data items and decisions $<D_1,a_1,D_2,a_2, ..., D_k,a_k,D_{k+1}>$, but now the path ends with a data item D_{k+1} for which no decision has been made. Given this information, the algorithm uses the choice function

$c_A^k : D^{k+1} \times H^k \to 2^H$ to provide a set of decisions $c_A^k(D_{1,} \dots, D_k, D_{k+1}, a_{1,} \dots, a_k)$. The node v will be extended with an edge for each decision a_{k+1} in this set. Instead of this, Borodin et al. in [1] define the decision function as $c_A^k : D^{k+1} \times H^k \to O(H \cup \{\perp\})$. In this definition the algorithm is allowed to give an order on the set of decisions returned by c_A^k . This is useful when we want to define the time of the computation to be of the size of the subtree to the left of the first optimal solution.

Definition 3. As defined in [1],[2]. A is a correct algorithm for a pBT search problem P if and only if for any YES instance I, T_A(I) contains at least one valid solution. For an optimization problem, the value of A(I) returned by the computation is the maximum/minimum value over all the valid solutions in the computation tree.

3.1 Multiple Orderings and/or Free Branches

Borodin et. al. [1] also define the concept of a free branch. We will both define this feature here and argue that it is equivalent to the more natural feature of allowing multiple orderings. Suppose, for example, the algorithm wants to have two branches. In one branch, it greedily goes though the data items using one order and in the other branch it greedily goes though the data items using a different order. The natural thing to do is to branch before the first data item is seen. Branching, however, is not allowed in the pBT model without seeing a data item first. More formally, when a pBT algorithm receives the kst data item D_k, it is allowed to use the decision function $c_A^{k-1}: D^k \times H^{k-1} \rightarrow 2^H$ to branch making more than one decision on this item. Then it can use the ordering function $r_A^k: D^k \times H^k \rightarrow O(D)$ to give one ordering for each of these branches. The algorithm, however, is not allowed to have multiple orderings. One way to fix the above problem is to instead allow our ordering function to return a set of orderings, namely $r_A^k: D^k \times H^k \rightarrow 2^{O(D)}$.

A second way to fix the problem is to use dummy data items, R_j. We will see that this fix feels indirect and ad hoc, but it is the one used in [1] and even more importantly, it will make doing the pBT reductions in Section 4 easier. The dummy data items R_j that are introduced do not exist in the actual problem that the pBT algorithm is solving and the decisions made on them do not affect the final solution. However, they do allow the pBT algorithm to branch without seeing an actual data item. An odd consequence of this is that it lets the pBT algorithm have different data orderings on these different branches. More formally, suppose that after making a decision a_k about D_k, the algorithm wants to branch making multiple orderings using $r_A^k : D^k \times H^k \rightarrow 2^{O(D)}$. However, because this is not allowed, it instead uses the ordering function $r_A^k : D^k \times H^k \rightarrow O(D \cup \{R_j\})$ to select a dummy item R_j. Then it uses the decision function $c_A^k : D^{k+1} \times H^k \rightarrow 2^H$ to branch on multiple dummy decisions about R_j. Finally, it uses the ordering function $r_A^{k+1} : D^{k+1} \times H^{k+1} \rightarrow O(D)$ to give one ordering for each of these dummy branches.

Sashka Davis in her PHD thesis, yet, provides another way to allow multiple orderings [8]. Her algorithm has two types of states: free-branching state and input-reading state. The free-branching state allows for multiple ordering and input reading state uses a priority function to order all the data items, and then reads the next data item from the input instance. Within this framework she obtained a $2^{\sqrt{n}}$ width lower bound for 7-SAT.

3.2 The Power of pBT Algorithms

The width 1 pBT algorithms capture many greedy algorithms such as Kruskal or Prim's algorithms for spanning tree, Dijkstra's shortest path algorithm, and Johnson's greedy 2-

approximation for vertex cover. To see how such problems fit into the pBT framework, let us simulate Kruskal's algorithm for finding a minimum spanning tree of a connected weighted graph as a width 1 pBT. To find the minimum spanning tree, we need to find a subset, S, of the edges that form a tree. S must include all the vertices in the graph. Also the total weight of all the edges in S should be minimized. Kruskal's algorithm maintains a list of all the edges in the graph. This list is sorted in increasing order of the edges' weights. The algorithm goes through every edge in the list one after another, if the revealed edge does not create a cycle then it is included in MST, otherwise it is discarded. This algorithm fits within the width 1 fixed order pBT framework. The list of edges is sorted once and the ordering is never changed again, which is why it is a fixed order algorithm. It is a width 1 pBT because, for each edge that the algorithm considers, it either accepts the edge or rejects it, but not both. An edge will get rejected only if adding it would create a cycle, therefore branching on accepting that edge would only make the solution inconsistent.

Another important class of algorithms that fit into this model are simple dynamic-programming or DP-simple. For example Knapsack, interval scheduling on m-machine and the string-edit-distance problem are known as DP-simple, and can be simulated with a fixed order pBT. For concreteness, lets look at the simulation of Knapsack within this framework. The Knapsack problem is defined as follows. We have a knapsack and a set of n items. Each item i has a value <p_i> and a weight <w_i>. Knapsack's capacity or the maximum weight that it can carry is C. We want to pick a subset of the items, so that the sum of their weights would be less than or equal to C and they would give us the maximum profit. The simple ½ -approximation algorithm, that solves this problem, fits within the adaptive order width 2 pBT. The way this algorithm works is that it first looks at the highest profit item and considers two decisions for this item, accept and reject. Therefore, this yields two branches in the computation tree. After that, for each branch, it sorts the items in decreasing order of their profit-to-weight ratio and greedily chooses items from the sorted list. Thus, each branch will have width 1, giving us a width 2 pBT. This algorithm is a "weak adaptive" one as it sorts the items only at two levels of the tree and not at every level; once at the beginning of the tree construction and then again right after seeing the first item. One of these solutions will be within a factor of 2 of the optimal solution.

Now let us see how this model simulates the dynamic programming algorithm for Knapsack in fixed pBT. This algorithm goes through the list of data items one at a time.

After seeing k data items $<D_1,...,D_k>$, the loop invariant is as follows. For each knapsack capacity $C' \leqslant C$, the algorithm stores in a table the solution to the sub-instance $(<D_1,...,D_k>,C')$, namely which of the items $D_1, ..., D_k$ seen so far to accept to optimally fill a knapsack with capacity C'. After reading the new data item D_{k+1} the dynamic programming algorithm maintains this loop invariant as follows. For each capacity $C' \leqslant C$, the algorithm considers both accepting and rejecting this data item. When it considers rejecting D_{k+1}, it extends the solution to the sub-instance $(<D_1,...,D_k>,C')$ stored in the table and when it considers accepting D_{k+1}, it extends the solution to $(<D_1,...,D_k>,C'-w_{k+1})$. Then it stores the best of these two solutions in the table indexed by $(<D_1,...,D_{k+1}>,C')$.

Within the fixed pBT framework, the algorithm, after seeing k data items $<D_1,...,D_k>$, stores this same solution for each knapsack capacity $C' \leqslant C$. The algorithm can either store these data items and the decisions made on them in the computation tree or as a set of partial solutions. As is required by the pBT model, the solution for $(<D_1,...,D_{k+1}>,C')$ will be an extension either of the solution for $(<D_1,...,D_k>,C')$ or for $(<D_1,...,D_k>,C'-w_{k+1})$. Either way, the new solution is an extension of a previously stored solution. It is interesting to see how pBT actually extends its computation tree. There is a path in the tree corresponding to the optimal solution for ($<D_1,...,D_k>$,C'), namely a_1 might say that D_1 is rejected while a_2 might say d_2 is accepted and so on. Then after reading D_{k+1} the algorithm is allowed to branch both accepting and rejecting it. The branch corresponding to rejecting D_{k+1}, will be a solution to the sub-instance ($<D_1,...,D_k,D_{k+1}>,C'$), while the branch corresponding to accepting D_{k+1}, will be a solution to the sub-instance ($<D_1,...,D_k,D_{k+1}>,C'+w_{k+1}$). The pBT algorithm will include these possible extensions only if they are optimal for these corresponding sub-instances. In the end, the solution corresponding to the sub-instance ($<D_1,...,D_n>,C$) will be the solution that gives the maximum profit.

As shown the model can capture many greedy and dynamic programming algorithms, yet the model has a restriction that prevents it from simulating all “backtracking” and “branch-and-bound” algorithms. The only way we can prune the tree for this model is by using the abort option, The decision whether to abort a path can depend only on the partial (observed) instances. This imposes a “locality” restriction on the model. In contrast, many branch-and-bound algorithms use a global pruning criterion (e.g., LP relaxation) and hence do not fit into the pBT model.

3.3 Results within the pBT Framework

In [1] the following three problems were studied : Interval Scheduling, Knapsack and Satisfiability. For all of them, lower bounds were proved within the pBT framework.

3.3.1 Interval Scheduling

In the interval scheduling problem, we have a set of intervals and each interval has a profit associated with it. We have to choose a subset of pairwise disjoint intervals so as to maximize the total profit.

This is the same as scheduling a set of jobs with time-intervals on one machine. If we have more than one machine then we must schedule jobs to machines so that there is no overlap between the jobs scheduled on any particular machine. The objective function is to maximize the overall profit of the scheduled jobs.

For m-machine interval Scheduling a tight (nm)- width lower bound in the adaptive-order pBT model, an inapproximability result in the fixed-order pBT model, and an approximability separation between width-1 pBT and width-2 pBT in the adaptive-order model was proved.

3.3.2 Knapsack

The knapsack problem has been defined previously. For optimality, it was shown that Knapsack requires an exponential width in the adaptive-order pBT. For achieving an FPTAS in the adaptive-order model an upper and lower bound that are polynomials in $1/\epsilon$ was proved. It was also shown that a width-1 pBT cannot approximate Knapsack better than a factor of $n^{-1/4}$ and that the standard ½-approximation is captured by width-2 pBT.

3.3.3 SAT

For the K-SAT problem, we have a boolean conjunctive-normal-form formula and the algorithm must return a satisfying assignment (if one exists). Note that each clause contains at most K propositional variables. For this problem two different models for representation of data items were considered (the weak and strong models). The representations differ in terms of the amount of information that is contained in the data items. The simplest weak data item representation includes a variable name along with the names of all the clauses that the variable is in. Also it indicates whether the variable occurs positively or negatively in each clause. For instance, the data item < x_i, (C_j,+), ($C_{j'}$,-) > means that x_i is contained in only two clauses C_j and $C_{j'}$ and that it appears positively in clause C_m, and negatively in clause C_n. In the strong model each data item specifies all the clauses that it is in and has information about the content of each clause. That is D_i =< x_i,C_1,C_2, ...,C_m > and C_1,C_2, ...,C_m are complete descriptions of the the clauses containing x_i.

For the SAT problem, it was proved that 2-SAT can be solved by a linear-width adaptive-order pBT, but requires exponential width for any fixed-order pBT. For MAX2SAT, a variation of 2SAT where we must find a satisfying assignment for as many clauses as possible, it was shown that no fixed-order pBT algorithm can approximate it efficiently. (Using a similar argument, an inapproximability result for Vertex Cover in the fixed-order pBT model was proved.) It was also proved that 3-SAT needs exponential width in the fully-adaptive-order pBT model.

3.3.4 3-SAT with Skip

In this section we introduce another model for 3-SAT, which we call 3-SAT with skip. To our knowledge, there has been no lower bounds proved for this model. Though it is not a very natural model in its own right, we believe if a lower bound could be proved for this model, then a wide range of problems such as Independent Set and Integer Multicommodity[2] could be reduced to it.

The first difference between the skip model and the standard model is that instead of a data item containing the full description of all the clauses that the current literal is in, it only describes one of these clauses. This of course means that the number of data items will be three times the number of clauses, instead of being the number of variables. For example $D_1 = <x_1, x_1 \vee x_2 \vee x_3>$, $D_2 = <x, x_1 \vee x_5 \vee x_6>$ and $D_3 = <x_1, \neg x_1 \vee x_4 \vee x_6>$ are three data items in this representation. The novelty of this model comes from the fact that we allow the 3-SAT algorithm to be more flexible in terms of making decisions on its data items. On receiving a new data item, $D_i = <x_j, C_k>$, the algorithm is no longer forced to either committing to x_j being true or to being false. Instead, it can also make the decision *skip*. When it does this, it has not committed to a value for x_j, but it has committed to the fact that clause C_k needs to be satisfied by a variable other than x_j. For example, it can later set x_j when it sees the data item $<x_j, C_{k'}>$ and it can ensure that the clause C_k is satisfied by setting $x_{j'}$ appropriately when it sees the data item $<x_{j'}, C_k>$.

2 The **Integer Multicommodity Problem** is a maximum-flow problem which involves multiple commodities. Each commodity has an associated demand and source-sink pairs. Note that the flow of each commodity through each edge must be an integer.

As done in Borodin et. al., the skip version of 3-SAT has both a strong and a weak version, where a data item in the strong representation contains the full description the clause, while the weak only includes the name of the clause.

4. BT Reduction

The BT-reduction between problems preserves the property of being efficiently computable by a BT algorithm. This reduction is very similar to NP-reduction.

If we are reducing problem A to problem B, then we need a mapping function, g, that will map the data items of problem A to (possibly sets of) data items for problem B. Also we need another mapping function, f, to map the decisions for problem B to decisions for problem A. From this reduction, we should be able to order the data items for A based on an ordering for the data items for B. Note that for this to happen, the sub-sets of data items of B corresponding to individual data items of A should be disjoint. Also once a decision is made on the first item in these subsets we should be able to use g to make a decision on the corresponding data item for problem A. In addition, we should be able to match the solution branch's of the computation tree, that is there should be a correspondence between the branches of B that represent a solution to the solution branches of A.

It is possible that some data items of B do not correspond to any A-item, therefore, we need to use some dummy data items. A formal definition of the BT-Reduction is as follows.

Definition 4. Let $P_1=\{\boldsymbol{D_1},\{f_1^n\}\}$ and $P_2=\{\boldsymbol{D_2},\{f_2^n\}\}$ to be two BT problems that use choices K_1 and K_2 respectively. A BT-reduction from P_1 to P_2 consists of the following:

- A function $g_1^n:\boldsymbol{D_1}\rightarrow P(\boldsymbol{D_2})$, such that for any two distinct $D, D' \in \boldsymbol{D_1}$, $g_1^n(D)\cap g_1^n(D')=\{\}$. This function maps each data items in P_1 to a set of data items in P_2. For two distinct data items in P_1 the corresponding sets in P_2 must not overlap.
- A function $g_2^n:\boldsymbol{D_2}\times K_2\rightarrow K_1$. This function maps the decisions made on the items in P_2 to decisions for data items in P_1.
- A set of items $\epsilon^n=\{E_1^n,\ldots,E_{m=m(n)}^n\in\boldsymbol{D_2}\}$ that are not in the image of g_1^n. These are the data items in $\boldsymbol{D_2}$ for which we need to introduce dummy data items

in P_1, because there are no data items in $\boldsymbol{D}_1$ that correspond to these data items.

These objects must satisfy the following property:

For any $I=\{D_1, \ldots, D_n\} \subset$ $\boldsymbol{D}_1$, let $g_1^n(I) \subset$ $\boldsymbol{D}_2$ denote the union of all images of the items in I under g_1^n. Let $\alpha : g_1^n(I) \rightarrow K_2$ be any assignment of decisions to these items. Finally for each D_i, let D_i' be any item in $g_1^n(D_i)$. Then,

$$f_1^n(D_1, \ldots, D_n, g_2^n(D'_1, \alpha(D'_1)), \ldots, g_2^n(D'_n, \alpha(D'_n))) = 1$$

if and only if

$$\exists b_1, \ldots, b_m \in K_2 \, f_2^{n'}(g_1^n(I), E_1^n, \ldots, E_m^n, \alpha(I), b_1, \ldots, b_m) = 1$$

This means that we will have a solution satisfying the objective function of P_1 *iff* we have a solution satisfying the objective function of P_2. P_1's objective function depends on the data items in P_1 and the choices made on those items. Here the choices for those data items is actually returned by our mapping function, which maps the decisions made on the corresponding items in P_2 to decisions for those data items in P_1. P_2's objective function considers both the data items in P_2 and the choices made on those items. The data item in P_2 are the union of the set of data items that are mapped from the data items in P_1 to some data items in P_2, $g_1^n(I)$, and $<E_1^n, \ldots, E_m^n>$, where $E_i^n s$ are the data items that are in P_2 but not in P_1. The set $<b_1, \ldots, b_m>$ are the decisions made about $<E_1^n, \ldots, E_m^n>$.

5. NP-Reductions

In this section, we look at some well-known NP-reductions and show whether or not they fit within the pBT framework. The reductions that we consider are as follows:

$$3-SAT \leq_p 0,1\ \textit{Integer Programming},$$
$$3-SAT \leq_p \text{Subset-Sum}/\textit{Knapsack}, \text{ and}$$
$$3-SAT \leq_p \textit{Independent Set.}$$

5.1 3-SAT Reduces to 0, 1 Integer Programming

In this section, we first show that 3-SAT reduces polynomially to 0,1 Integer Programming ($3-SAT \leq_p 0,1\ \textit{Integer Programming}$) through an NP reduction, and then we prove that this reduction is a BT-reduction. In the remainder of this section, we refer to 0,1 Integer Programming as IP.

5.1.1 NP-Reduction from 3-SAT to IP [11]

$3 - SAT \leq_p 0,1\, IP$

IP problem:

***Instance**: A set V of integer variables, a set of inequalities over V, a maximization function f(V), and an integer B.*

***Solution**: Does there exist an assignment of integers to V such that all inequalities are true and f(V) ≥ B?*

To show that 3-SAT reduces polynomially to IP, the first thing we need to do is to come up with a mapping function that maps the input instance of the 3-SAT problem (a set of clauses that contain propositional variables) to an input instance for IP. To do this, we will have twice as many variables in IP, one for each positive occurrence of the literals and one for each negative occurrence. We will also have the following inequalities:

- $0 \leq v_i \leq 1;$
- $0 \leq \neg v_i \leq 1;$
- $1 \leq v_i + \neg v_i \leq 1;$

- For each clause C_i we will have $v_i+v_i'+ v_i''\geq 1$. (Note that if the clause contains a variable in the negative form then we will have something like $v_i+\neg v_i'+v_i''\geq 1$.); and
- $B = 0$.

Now we need to have a mapping between the solutions of the two problems. We map every true literal to 1 in the integer programming and the false literal to 0 and vice versa.

We need to show that if there is satisfying assignment for the 3-SAT problem then there is solution for the IP problem. We know that 3-SAT is satisfied, that means at least one literal per clause is true, so the inequality sum for every clause is maintained. In addition since 3-SAT has a valid solution, for every literal $1\leq v_i + \neg v_i \leq 1$ must hold.

We also need to show that if there is solution for our IP then 3-SAT is satisfied. In the solution for the IP, every variable will be 0 or 1. So we will set the literals corresponding to 1 as true and 0 as false. Since no boolean variable and its complement can both be true, because of $1\leq v_i + \neg v_i \leq 1$, we will have a legal assignment that also must satisfy every clause (since every clause inequality must hold).

5.1.2 NP-Reduction from 3-SAT to IP is a BT-Reduction

$$3-SAT\ strong \leqslant_{BT} 0{,}1\ Integer\ Programming$$

In the previous section we gave an NP-reduction from 3-SAT to IP, in this section we want to show that this reduction is a BT reduction. That is we want to prove that if we have a pBT algorithm for the IP problem, we can build a pBT algorithm for 3-SAT.

We defined the 3-SAT and the IP problems previously, yet once we are working in the pBT framework we need to have a definition for the data items in these problems. Each 3-SAT data item in the strong representation consists of a variable with the full description of all the clauses that the variable appears in. For example, $D_i = <x_i, (x_i \vee x_j \vee x_k), (\neg x_i \vee x_j \vee x_k)>$. Each data item in IP contains the name of the variable along with the full description of all the inequalities that the variable is in. The two IP data items corresponding to D_i are $D_{<i,+>} = <x_{<i,+>}, (x_{<i,+>}+x_{<j,+>}+ x_{<k,+>} \geq 1), (1\leq x_{<i,+>}+ x_{<i,->} \leq 1), (0 \leq x_{<i,+>}\leq 1)>$ *and* $D_{<i,->} = <x_{<i,->},(x_{<i,->}+x_{<j,+>}+x_{<k,+>}\geq 1), (1\leq x_{<i,+>}+x_{<i,->} \leq 1), (0 \leq x_{<i,-}\leq 1)>$.

In order to describe the algorithm for 3-SAT, let us leap into the middle of the computation and state what our loop invariant is (since the algorithm works recursively, you might as well call this a precondition as opposed to loop invariant). We have built a partial computation tree for 3-SAT and we are at currently considering extending a particular leaf edge e. The nodes along the path from the root to this current edge e is labeled with the data items <D_1, ..., D_k> seen so far and the edges extending these nodes are labeled with the decisions <a_1, ..., a_k> made about them (figure 3a). We have also built a mirror computation tree for IP. The corresponding path in the computation tree for IP is illustrated in figure 3b.

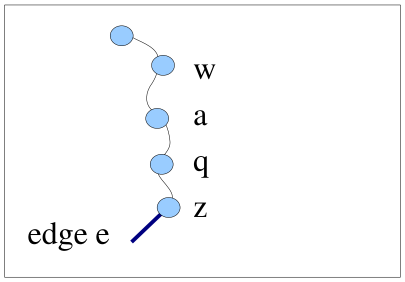

Figure 3a. A path in the computation Tree for 3-SAT.

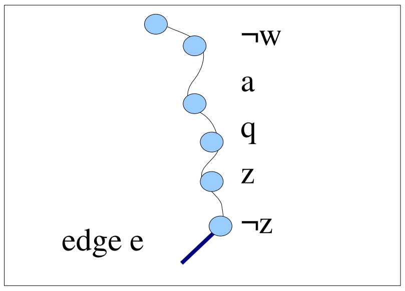

Figure 3b. A path in the computation Tree for IP.

Also at this point we have a set, $P_{e,}$ of 3-SAT data items that are possibly the next to be seen. Let $P'_{<e\ \pm>}$ = $\{D_{<i,+>}, D_{<i,->} | D_i \in P_e\}$ denote the set of IP data items mapped to the 3-SAT data items in P_e. These data items are possible future data items for IP. There is also a set referred to as *known-set*, P'_e, consisting of additional data items that we know exist in the IP's input. We have this knowledge because we have already seen either the positive or negative form of that data item. For example, if we look at figure 3b, we know w, ¬a and ¬q exists for IP because IP has already seen ¬w, a and q. The set of next possible data items for IP is $P_{<e, \pm>} \cup P'_e$.

We need to get the next data item for 3-SAT. This is accomplished by ordering 3-SAT's set of all possible data items, and picking the one that comes first in this ordering. Formally, we need to define ordering function $r_A^k : D^k \times H^k \rightarrow O(D)$. Given that so far we have seen the data items $<D_1, ..., D_k>$ and made the decisions $<a_1, ..., a_k>$, our current goal is to provide the order $r_A^k(D_1, ..., D_k, a_1, ..., a_k)$ for the set of possible future data items in P_e. Let us first consider the obvious, yet faulty way of defining an ordering for 3-SAT data items based on an ordering for IP's data items.

The formal definition of the reduction defines the ordering of 3-SAT's data items by first defining a function $g_1^n : D_1 \rightarrow P(D_2)$. Here g_1^n will map each 3-SAT data item D_i to the set $\{D_{<i,+>}, D_{<i,->}\}$. This induces the obvious ordering $r_A^k(D_1, ..., D_k, a_1, ..., a_k)$ of P_e from the ordering that IP gives to data items in $P_{<e \pm>} = \{D_{<i,+>}, D_{<i,->} | D_i \in P_e\}$. We put D_i before D_j in our ordering, if IP puts either $D_{<i,->}$ *or* $D_{<i,+>}$ first of the four items $\{D_{<i,+>}, D_{<i,->}, D_{<j,+>}, D_{<j,->}\}$. Suppose, as an example, that the remaining possible data items for us is $P_e = \{D_4, D_9, D_{10}\}$. IP's corresponding possible future items are $P_{<e, \pm>} =$

$\{D_{<4,+>}, D_{<4,->}, D_{<9,+>}, D_{<9,->}, D_{<10,+>}, D_{<10,->}\}$. In addition, suppose IP's set of possible future items includes the *known set* $P'_e = \{D_{<8,+>}, D_{<13,->}\}$. The order that IP gives $P_{<e, \pm>} \cup P'_e$ might be $(D_{<4,+>}, D_{<8,+>}, D_{<4,->}, D_{<13,->}, D_{<9,+>}, D_{<10,->}, D_{<10,+>}, D_{<9,->})$. We will define from this our ordering (D_4, D_9, D_{10}) in 3-SAT. If D_4 is in 3-SAT's actual input instance, then the ordering works fine. We will see D_4 next and IP sees $D_{<4,+>}$ next. A problem arises when our next data item in IP's ordering comes after a data item from P'_e. For example, suppose our instance does not contain D_4 but does contain D_9 and D_{10}. From our ordering $\{D_4, D_9, D_{10}\}$, we should get D_9 before D_{10}. However, this might not correctly mirror IP's computation. In this case, IP sees $D_{<8,+>}$ next. We won't see D_8, because we saw it when IP saw $D_{<8,->}$. After seeing $D_{<8,+>}$ IP, being adaptive, might change its ordering. For example, it might decide to reorder its remaining possible data items as follows: $(D_{<13,->}, D_{<9,->}, D_{<9,+>}, D_{<10,+>}, D_{<10,->})$. Then IP sees $D_{<13,->}$ next, which won't see either. After seeing this IP might change its ordering yet again, to say $(D_{<10,->}, D_{<9,+>}, D_{<10,+>}, D_{<9,->})$. IP next sees $D_{<10,->}$. If we are going to mirror its computation, then we should see D_{10} and not D_9 next. The complication for us is that we need to define our complete ordering P_e before we know which data items are actually in our input instance. We instead define our ordering using an iterative algorithm. To come up with this algorithm, we first assume a particular scenario and then show that it works in general as well.

Let us suppose that we in 3SAT know we have seen all the data items in our actual input instance, i.e. none of the possible future data items, P_e, are in the actual input instance. IP does not have this information so it will order the items in the set $P'_{<e, \pm>}$ as well as the ones in P'_e. Let $D_{<i1,+>}$ denote the first data item in this ordering from P'_e. Consider the prefix O'_1 of this ordering that comes before this data item $D_{<i1,+>}$. This prefix will impose an ordering on some of the data items in P_e. Lets call this partial ordering O_1. After IP sees $D_{<i1,+>}$ it might change its ordering. As a loop invariant, suppose that IP has seen j data items from P'_e, denoted $D_{<i1,+>}$, $D_{<i2,->}$, ..., $D_{<ij,+>}$ and before each of these we mirrored IP's ordering on $P'_{<e, \pm>}$ to define for us the partial orders O_1, O_2, ..., O_j. IP has learned that the data items in $O'_1 \cup O'_2 \cup \ldots \cup O'_j$ are not in the input instance. So at this point IP sees the data item $D_{<ij,+>} \in P'_e$. An extra challenge for us is that IP may branch in its computation tree making multiple decisions on $D_{<ij,+>}$. Luckily we know that at most one of these leads to a valid solution for IP. Let us assume that IP decided previously to set $D_{<ij,->}$'s variable $x_{<ij,+>}$ to one. Now as IP sees $D_{<ij,+>}$ we know that for IP to succeed, it needs to set $x_{<j,->}$ to zero. The IP algorithm may or may not be smart enough to do this. Using our extra knowledge, we ignore IP's inconsistent branch. If IP aborts or does not have a consistent branch then we will want to abort this path in our computation tree. We will later describe how we do this. Otherwise, we follow IP's consistent branch. Following this branch, IP will order the remaining items $P'_{<e,\ \pm>} \cup P'_e$. Let $D_{<i(j+1),+>}$ denote the first data item in this ordering from P'_e. Consider the prefix O'_{j+1} of this ordering that comes before this data item $D_{<i,+>}$. This prefix will impose an ordering on some of the data items in P_e. Lets call this ordering O_{j+1}. As an extra note, we remove from O_{j+1} any of our data items included earlier in O_1 O_2 ... O_{j+1}. Suppose, as in the example above, that IP has learned that the data item $D_{<4,+>}$ is not in its instance because IP put this data item in its ordering O'_1 before $D_{<i1,+>}$, but IP saw $D_{<i1,+>}$ first. With a little extra wisdom from us, IP would also know that the compliment data item $D_{<4,->}$ is also not in its instance. With out this wisdom, however, it might put $D_{<4,->}$ in the ordering O'_{j+1}. We knowing better, would not include D_4 in the ordering O_{j+1}, because D_4 is already in our ordering O_1.

We continue doing this until IP has no more data items left. Our ordering will be the concatenation of these partial orderings O_1 O_2 ... O_J. This process is visualized in figure 4.

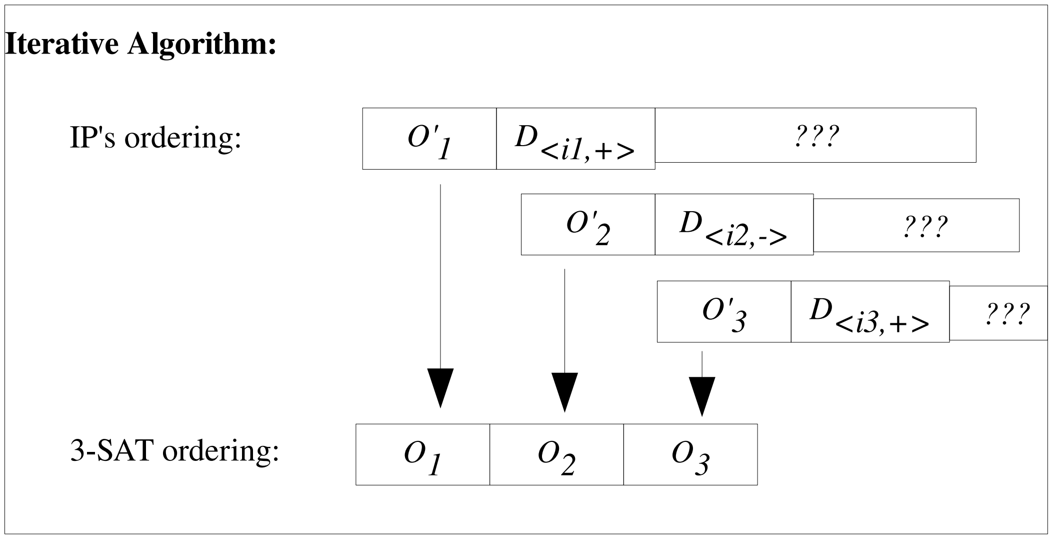

Figure 4. The iterative algorithm to induce an ordering for 3-SAT data items. The data items in O'_i are from $P'_{<e, \pm>}$, while the ones in O_i are from P_e.

Defining our ordering in this way works in the general case as well. Suppose our actual input instance does not contain any data items in $O_1 \cup O_2 \cup \ldots \cup O_j$, but does contain D_i from O_{j+1}. Then IP's input instance does not have any from $O'_1 \cup O'_2 \cup \ldots \cup O'_j$. IP first sees $D_{<i1,+>}$, $D_{<i2,->}$, ..., $D_{<ij,+>}$ from P'_e and reorders its possible data items after each of these as defined above. IP next sees $D_{<i,+>}$ according to the ordering O'_{j+1} and we next see D_i, according to the ordering O_{j+1}.

Using the above iterative algorithm we get an ordering for our data items. We will next see D_{k+1} (denoted above by D_i), which is the first data item in our input instance according to this ordering. Our next step is to make a decision about D_{k+1}. Formally, we need to define choice function $c^k_A : D^{k+1} \times H^k \rightarrow 2^H$. Given that so far we have seen the data items <D_1, ..., D_k> and made the decisions <a_1, ..., a_k> and now have seen D_{k+1}, our current goal is to provide the set of decisions $c^k_A(D_1, \ldots, D_k, D_{k+1}, a_1, \ldots, a_k)$ about D_{k+1}. Suppose that IP sees $D_{<k+1,->}$ and sets $x_{<k+1,->}$ to one. By the NP-reduction IP variable $x_{<k+1,->}$ corresponds to the negation of the 3-SAT variable x_{k+1}. Hence, when we follow the IP algorithm's lead we set x_{k+1} to false. The newly seen data item D_{k+1} is added as a new node to our path in our computation tree and the new decision false is added as a new edge. Given that we know that IP will eventually see its matching data item $D_{<k+1,+>}$, we complete this iteration by putting this $D_{<k+1,+>}$ into IP's set *known set* P'_e.

If, on the other hand, the IP algorithm sets $x_{<k+1,->}$ to zero, then we do the same as above except we set x_{k+1} to true. If IP aborts, then we abort. If IP branches with both the decisions to set $x_{<k+1,->}$ to zero and to one, then we too branch setting x_{k+1} to both true and false. When two decisions are made we have to “recursively” start the next iteration on both of the two extensions.

There is yet one more complication. When IP sees a data item $D_{<ij,+>}$ from its *known set* P'_e, it may not have a consistent branch or it may decided to abort its current branch. In either case, we would like to abort our current branch as well but the model does not allow us to do this. We cannot abort without seeing any data items first. One way to fix this is as follows. We have defined the partial ordering $O_1, O_2, ..., O_j$ already. Because IP is aborting, we cannot define the rest of the order $O_{j+1}, ..., O_J$ as done above. But the rest of the order will not matter, so we order these remaining data items in an arbitrary way. Then, when we see our next data item D_{k+1}, if it is from $O_1, O_2, ..., O_j$, then we make a decision as described above. On the other hand, if it is from $O_{j+1}, ..., O_J$ then we abort.

We will continue with our algorithm till there are no more new data items left for us to consider. Throughout the construction of our computation tree the loop invariant is maintained. Hence, our new 3-SAT algorithm directly parallels the IP algorithm. The NP-reduction gives that we find a satisfying assignment to the given 3-SAT input instance if and only if the IP algorithm does so. Our 3-SAT computation tree is no larger than IP's computation tree. Hence, if we have a small width algorithm that solves IP in the pBT framework, then we can come up with a small width pBT algorithm that solves 3-SAT. From [1] we already know that 3-SAT needs an exponential width in the fully-adaptive-order pBT model. Therefore any fully adaptive algorithm solving IP in this framework requires an exponential width as well.

5.2 3-SAT Reduces to Knapsack/Subset-Sum

In this section, we present an NP-reduction from 3-SAT to Knapsack/Subset-sum and show how this reduction can be made a BT-reduction when the BT model includes free branching.

5.2.1 NP-Reduction from 3-SAT to Knapsack/Subset-Sum [5]

$3-SAT \leqslant_p Knapsack$

Subset-Sum Problem:

Instance: *A set of positive integers, S, and a target t.*

Solution: A *subset of S that sums up to t.*

The reduction from 3-SAT to Subset-Sum is as follows. First we need to build an instance for the Subset-Sum problem from the instance to the 3-SAT problem. To do this, for each variable x_i in 3-SAT we will create two base 10 integers $D_{<i,+>}$ and $D_{<i,->}$ in Subset-Sum. The digits of one integer specifies the variable and which clauses use the positive form of the literal in 3-SAT while the digits in the second integer specifies which clauses use the negative form. In addition, for every clause in 3-SAT we will introduce 2 clause integers in subset-sum. One will have 1 in the column corresponding to that clause and 0 every where else and another one will have 2 in the column corresponding to that clause and 0 every where else. The target will have digit 1 in the columns corresponding to variable names and digit 4 in the columns corresponding to clauses. Table 1 demonstrates how this transformation is done for the 3-SAT instance $(x_1 \lor \neg x_3 \lor \neg x_4) \land (\neg x_1 \lor x_2 \lor \neg x_4)$.

	x_1	x_2	x_3	x_4	Clause C_1	Clause C_2
$D_{<1,+>}$	1	0	0	0	1	0
$D_{<1,->}$	1	0	0	0	0	1
$D_{<2,+>}$	0	1	0	0	0	1
$D_{<2,->}$	0	1	0	0	0	0
$D_{<3,+>}$	0	0	1	0	0	0
$D_{<3,>}$	0	0	1	0	1	0
$D_{<4,+>}$	0	0	0	1	0	0
$D_{<4,->}$	0	0	0	1	1	1
$D_{<C1,1>}$	0	0	0	0	1	0
$D_{<C1,2>}$	0	0	0	0	2	0
$D_{<C2,1>}$	0	0	0	0	0	1
$D_{<C2,2>}$	0	0	0	0	0	2
t	1	1	1	1	4	4

Table2. Transforming $(x_1 \lor \neg x_3 \lor \neg x_4) \land (\neg x_1 \lor x_2 \lor \neg x_4)$ to a Subset-Sum instance.

We also need to map each Subset-Sum solution to 3-SAT a solution. If T_i is included in the solution subset, we set the corresponding variable x_i to true in 3-SAT and false, otherwise. Now we need to show that if the Subset-Sum solution is valid and sums to t, then 3-SAT is satisfied and has a valid solution. For each 3-SAT variable x_i, the digit of the target in the corresponding column is a one. Hence, exactly one of $D_{<i,+>}$ and $D_{<i,->}$ is included in the Subset-Sum solution. Therefore, the 3-SAT assignment is defined in a consistent way. For each 3-SAT clause C_j, we need to include at least one integer with a one in the column corresponding to this clause. This included integer will correspond to a 3-SAT literal that satisfies the clause.

Conversely, we also need to map each 3-SAT solution to a Subset-Sum solution. If x_i is set to true, then the corresponding integer $D_{<i,+>}$ is included in the Subset-Sum solution and $D_{<i,->}$ is not. Conversely, if x_i is set to false, then $D_{<i,->}$ is included and $D_{<i,+>}$ is not. Also for each 3-SAT clause C_j , if only one literal satisfies the clause then we include both $D_{<Cj,1>}$ and $D_{<Cj,2>}$. If two literal satisfies the clause then we include only $D_{<Cj,1>}$. If three, then we include neither $D_{<Cj,1>}$ nor $D_{<Cj,2>}$. It is fairly obvious that if this 3-SAT solution satisfies our 3-SAT instance, then each column corresponding to a variable sums to one and each column corresponding to a clause sums to four and hence this Subset-Sum solution correctly sums to the target t.

5.2.2 NP-Reduction from 3-SAT to Knapsack/Subset-Sum is a BT-Reduction

In this section we will prove $3-SAT\ weak \leq_{BT} \text{Subset-Sum}$ through a BT reduction, when the BT model includes free branching.

We defined the 3-SAT and the Subset-Sum problems previously, yet once we are working in the pBT framework we need to have a definition for the data items in these problems. Each 3-SAT data item in the weak representation contains a variable x_i along with the names of the clauses that literal is in including whether it appear positively or negatively. For example, $D_i = < x_i, (C_j,+), (C_{j'},-) >$. (In contrast, data items in the strong representation includes complete description of these clauses.) Each data item in Subset-Sum consists simply of a single integer from the input instance. The target integer t is either globally known or is a data item that our algorithm will read first. The goal is to find a subset of the input data items that sum up to target t.

Our desired result $3-SAT\ weak \leq \text{Subset-Sum}$ is stronger than the result $3-SAT\ strong \leq \text{Subset-Sum}$ because if we have an algorithm solving 3-SAT weak we can come up with an algorithm for 3-SAT strong and hence $3-SAT\ strong \leq 3-SAT\ weak$.

Our BT reduction from weak 3-SAT to Subset-Sum will be identical to that for IP in Section 4.1.2 except with a few additional complications that we will point out here. In the IP reduction the IP data items mapped to the 3-SAT data item D_i were denoted $D_{<i,+>}$ and $D_{<i,->}$. Note that we do the same here, except that here $D_{<i,+>}$ and $D_{<i,->}$ are integers as defined above. Just as in the IP reduction, we define P_e, to be the set of possible future 3-SAT data items and $P'_{<e\ \pm>} = \{D_{<i,+>}, D_{<i,->} \mid D_i \in P_e\}$ to be the corresponding Subset-Sum data items. As before, P'_e refers to the *known-set* consisting of the Subset-Sum data items $D_{<i,+>}$ after its matching item $D_{<i,->}$ has already been seen. The key difference is that Subset-Sum also has the clause data item integers $D_{<Cj,q>}$. The clause data items that are still to be seen will be included in the set P'_e. With these same definitions, the reduction for Subset-Sum goes through the same as that for IP. As before, when Subset-Sum makes multiple decisions about a data item $D_{<i,+>}$ in P'_e we know which branch will lead to a valid solution, namely the complement of the answer that Subset-Sum already gave for $D_{<i,->}$. However, a complication is when Subset-Sum makes multiple decisions about a clause data item $D_{<Cj,q>}$ in P'_e. In this case, we no longer know which branch will lead to a valid solution. Hence, we need to branch making both of these decisions. The complication is not caused by the fact that Subset-Sum both accepts $D_{<Cj,q>}$ in one path and rejects it in another. The complication arises when Subset-Sum orders its possible future data items differently in these two branches. The formal definition of the fully adaptive pBT model allows branching in order to make multiple decisions about a data item just seen, but does not allow branching in order to provide multiple orderings for selecting the next data item to be seen. We in 3-SAT do not see a data item corresponding to this Subset-Sum data item $D_{<Cj,q>}$ and hence we are not allowed to branch yet and specify these two different orderings. One could solve this problem by extending the BT model to allow free branches, which would allow 3-SAT to branch with multiple orderings (See Section 2.1). However, this solution is hard to explain when Subset-Sum sees many data items from P'_e. We will instead consider the equivalent extension of the BT model that allows dummy data items. When Subset-Sum sees a clause data item $D_{<Cj,q>}$, 3-SAT orders its future data items so that it sees a corresponding dummy data item. More formally, suppose that Subset-Sum has seen j data items from P'_e, denoted $D_{<i1,+>}$, $D_{<i2,->}$, ..., $D_{<i(j-1),+>}$, $D_{<Cj,q>}$ and before each of these we mirrored Subset-Sum's ordering on $P'_{<e,\pm>}$ to define for us the partial orders $O_1, O_2, ..., O_j$. Here $D_{<Cj,q>}$ is the first clause data item on which multiple decisions are made. We append our partial ordering $O_1, O_2, ..., O_j$ with a dummy data item and order the remaining data items after this arbitrarily. The proof of correctness of this ordering is the same as IP for the $O_1, O_2, ..., O_j$ part of the ordering. However, if our actual input instance does not contain any data items in $O_1 \cup O_2 \cup ... \cup O_j$, then Subset-Sum's input instance does not have any item from $O'_1 \cup O'_2 \cup ... \cup O'_{j'}$. Subset-Sum first sees $D_{<i1,+>}$, $D_{<i2,->}$, ..., $D_{<i(j-1)+>}$ from P'_e and reorders its possible data items after each of these as defined above. Subset-Sum next sees $D_{<Cj,q>}$ and we next see the corresponding dummy data item. Subset-Sum branches in order to make multiple decision for $D_{<Cj,q>}$ and we branch in order to make multiple decision for this dummy. We, however, don't really care about the different decisions made about this dummy item. The different branches will allow us to follow Subset-Sum's different paths and, if necessary, to have different future orderings on the data items.

The rest of the proof goes through the same. Hence, our new 3-SAT algorithm directly parallels the Subset-Sum algorithm. The NP-reduction gives that we find a satisfying assignment to the given 3-SAT input instance if and only if the Subset-Sum algorithm does. Our 3-SAT computation tree is no larger than the Subset-Sum computation tree. Hence, if we have a small width algorithm that solves Subset-Sum in the pBT, then we can come up with a small width pBT with free branching algorithm that solves 3-SAT.

[2] proves that 3-SAT needs exponential width in the fully-adaptive-order pBT model, but their proof does not allow free branching. [8] extends their result allowing for free branching, except in [8] a $2^{\sqrt{n}}$ width lower bound is obtained for 7-SAT instead of 3-SAT. The above reduction could easily be changed to be a reduction from 7-SAT instead of from 3-SAT. Therefore, any fully adaptive algorithm solving Subset-Sum in this framework requires $2^{\sqrt{n}}$ width as well.

5.3 3-SAT Reduces to Independent Set (Ind. Set)

In this section we will present an NP-reduction from 3-SAT to Independent Set and show Ind. Set reduces from 3-SAT with skip through a BT reduction.

5.3.1 NP-Reduction from 3-SAT to Independent Set (Ind. Set) [10]

$3 - SAT \leqslant_p Ind.\,Set$

Ind. Set Problem:

Instance: *Graph G = (V, E,t) where V is the set of vertices and E is the set of edges.*

Solution: *An independent set of vertices, i.e. a subset V’ of V of size t such that there are no edges joining any two vertices of V’.*

A common NP-reduction from 3-SAT to the independent set problem is as follows. For every clause C_j, it defines a clause triangle. Each vertex in the clause triangle corresponds to a literal in the clause C_j. Then for every variable x_i, it adds an edge between each vertex labeled x_i and each vertex labeled with its complement $\neg x_i$. This transformation is illustrated in figure 5. Clearly the size of G is polynomial in the number of clauses and variables.

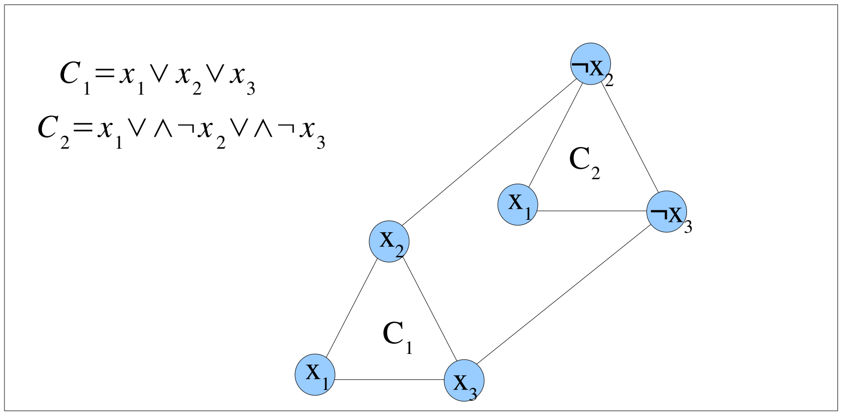
$C_1 = x_1 \vee x_2 \vee x_3$
$C_2 = x_1 \vee \wedge \neg x_2 \vee \wedge \neg x_3$
$\neg x_2$
C_2
x_1
$\neg x_3$
x_2
C_1
x_1
x_3

Figure 5. Constructing the graph for the IS problem from a set of 3-SAT .

We need to show that 3- SAT instance is satisfiable *iff* G contains an independent set of size m, where m is the number of clauses. Any independent set, I, in graph G must contain one vertex per clause triangle. Also both x_i and $\neg x_i$ cannot be in I, since otherwise it would not be an independent set. Assigning true to all the literals corresponding to vertices in the independent set will give us a valid solution for 3-SAT . Also when 3- SAT has a solution, we know that at least one literal per clause is satisfied. Therefore for every clause, C, we will include in our independent set the node corresponding to one of the satisfied literals in C. This will give us a valid independent set of size m.

5.3.2 NP-Reduction from 3-SAT with Skip to Ind. Set is a BT-Reduction

$$3-SAT\ strong\ with\ Skip \leqslant_{BT} Ind.\ Set$$

In the previous section we gave an NP-reduction from 3-SAT to Ind. Set, in this section we want to show that this reduction is a BT reduction from 3-SAT with skip. That is we want to prove that if we have a pBT algorithm for the Ind. Set problem, we can build a pBT algorithm for 3-SAT with skip.

As with the previous BT-reductions the first thing we need to do is to define the data items for each problem.

For 3- SAT with skip each data item contains a variable and full description of one of the clauses that the variable is in, e.g. $D_{<xi,Cj>} = <x_i, C_j> = <x_i, (\neg x_i \vee x_{i'} \vee x_{i''})>$. In Ind. Set each data item is a node that knows the name of its neighbours. For example, clause $C_1 = (\neg x_i \vee x_{i'} \vee x_{i''})$ in 3-SAT is mapped to nodes for each of its three literals. The node corresponding to the literal $\neg x_i$ is called $D_{<xi,Cj>}$.

Our BT reduction from 3-SAT to Independent Set will be much easier than that for IP in Section 4.1.2. In the IP reduction, two IP data items $D_{<i,+>}$ and $D_{<i,->}$ are mapped to each 3-SAT data item D_i. Hence, when defining the set of possible future data items for IP, we need to define the sets $P'_{<e\ \pm>} = \{D_{<i,+>}, D_{<i,->} | D_i \in P_e\}$ and P'_e. Now, however, we have a one-to-one mapping between 3-SAT's data items $D_{<xi,Cj>}$ and Independent set's data items.

The complication in the reduction comes from the fact that Ind. Set sees many different nodes corresponding to the same variable x_i in either of the strong or weak representation. Wanting consistent value for this variable, 3-SAT needs to be careful with Ind. Set accepting some of these nodes and rejecting others.

When Ind. Set sees the data item $D_{<xi,Cj>}$ we see the corresponding 3-SAT data item $D_{<xi,Cj>}$. If Ind. Set accepts $D_{<xi,Cj>}$, then we have no problem because we know to set the variable x_i to have the value that satisfies the clause C_j. If, on the other hand, the Ind. Set rejects $D_{<xi,Cj>}$, then we do not know how to set the variable x_i. All we know is that one of the other two nodes in the clause C_j triangle must be accepted. Hence, we know that one of the other two literals in clause C_j must satisfy C_j. This motivated the definition of the skip model in which the algorithm is no longer forced to either committing to x_i being true or to being false. Instead, it can also make the decision *skip*. When it does this, it has not committed a value for x_i, but it has committed to the fact that clause C_j needs to be satisfied by a variable other than x_i. For example, it can later set x_i when it sees the data item $<x_i, C_k>$ and it can ensure that the clause C_j is satisfied by setting $x_{i'}$ appropriately when it sees the data item $<x_{i'}, C_j>$.

At this point this reduction is complete. No lower bounds have been proved for the 3-SAT with skip model, therefore we cannot say anything about the required width of the Ind. Set's computation tree.

6. Conclusion and Future Works

In this project we presented the pBT and pBT with free branching frameworks. The latter model allows us to have multiple orderings. We also included the results obtained within these frameworks for well-known problems (i.e., 3-SAT , Knapsack, Interval Scheduling and 7-SAT). We proved a lower bound for 0,1 Integer Programming and Sunset-Sum/Knapsack by reducing them to 3-SAT and 7-SAT[3], respectively. We also provided a direction to further extend this model, namely 3-SAT with skip. Our extension allows the

3 Note that we actually presented a reduction from 3-SAT to Subset-Sum but it is quite obvious that the reduction can be easily modified to be a reduction from 7-SAT to Subset-Sum.

model to be more flexible in terms of making decisions. We believe this extension increases the extensiveness of the model. To demonstrate a situation where this extension is useful, we showed that Independent Set reduces from 3-SAT with skip. Therefore we believe proving a lower bound for this 3-SAT model would be quite interesting.

References:

[1] M. Alekhnovich, A. Borodin, J. Buresh-Oppenheim, R. Impagliazzo, A. Magen, and T. Pitassi. Toward a model for backtracking and dynamic programming. In *IEEE Conference on Computational Complexity,* pages 308-322, 2005.

[2] M. Alekhnovich, A. Borodin, J.Buresh-Oppenheim, R. Impagliazzo, A. Magen, and T. Pitassi. Toward a model for backtracking and dynamic programming. 2008.

[3] S. Angelopoulos and A. Borodin. On the power of priority algorithms for facility location and set cover. *Algorithmica*, 40(4):pp.271-291, 2004.

[4] A. Borodin, M. N. Nielsen, and C. Rackoff. (Incremental) Priority Algorithms. *Algorithmica*, 37(4):pp.295-326, 2003.

[5] T. Cormen, C. Leiserson, R. Rivest and C. Stein. *Introduction to Algorithms, Second Edition (Page 1015).* MIT Press, Cambridge, Mass., 2001.

[6] S. Davis and R. Impagliazzo. Models of greedy algorithms for graph problems. In *SODA, New Orleans, Louisiana, USA*, pages 381-390. SIAM, 2004.

[7] S. Davis. Evaluating algorithmic design paradigms. In *Grace Hopper Celebration of Women in Computing,* 2006.

[8] S. Davis PHD thesis, Department of Computer Science and Engineering, University of California, San Diego, pages 47-69, 2008.

[9] J. Edmonds, *How to Think about Algorithms. First Edition.* Cambridege University Press. Cambridge, 2008.

[10]J. Kleinberg, E. Tardos. *Algorithm Design, First Edition*. Addison-Wesley, 2006.

[11] Steve S. Skiena, *The Algorithm Design Manual, First Edition (Page 147).*Springer-Verlag New York, 1998.